Table of Contents

Elasticsearch Java API 手册



本手册由 全科 翻译，并且整理成电子书，支持PDF,ePub,Mobi格式，方便大家下载阅读。

阅读地址：http://woquanke.com/books/esjava/

下载地址：https://www.gitbook.com/book/quanke/elasticsearch-java

github地址：https://github.com/quanke/elasticsearch-java

gitee 地址：https://gitee.com/quanke/elasticsearch-java

配套示例代码：https://gitee.com/quanke/elasticsearch-java-study

编辑：http://woquanke.com

编辑整理辛苦，还望大神们点一下star ，抚平我虚荣的心

> 不只是官方文档的翻译，还包含使用实例，包含我们使用踩过的坑

推荐阅读

Elasticsearch Java Rest 手册 已经完成大部分

更多请关注我的微信公众号：

下面几个章节应用的相对少，所以会延后更新，计划先把 配套实例 elasticsearch-java-study 项目写完；

- Indexed Scripts API
    - Script Language
- Java API Administration
    - Indices Administration
    - Cluster Administration

参考

- elasticsearch java API 官方文档
- elasticsearch性能调优
- ElasticSearch 5.0.1 java API操作
- fendo Elasticsearch 类目
- Java API 之 滚动搜索(Scroll API)

安装

Maven Repository

Elasticsearch Java API包已经上传到 Maven Central

在 `pom.xml` 文件中增加：

> transport 版本号最好就是与Elasticsearch版本号一致。

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>transport</artifactId>
    <version>5.6.3</version>
</dependency>
```

Client

Java 客户端连接 Elasticsearch

一个是 `TransportClient` ，一个是 `NodeClient` ，还有一个 `XPackTransportClient`

- TransportClient：

作为一个外部访问者，请求ES的集群，对于集群而言，它是一个外部因素。

- NodeClient

作为ES集群的一个节点，它是ES中的一环，其他的节点对它是感知的。

- XPackTransportClient：

服务安装了 `x-pack` 插件

> 重要：客户端版本应该和服务端版本保持一致
>
> TransportClient旨在被Java高级REST客户端取代，该客户端执行HTTP请求而不是序列化的Java请求。 在即将到来的Elasticsearch版本中将不赞成使用TransportClient，建议使用Java高级REST客户端。
>
> 上面的警告比较尴尬，但是在 5xx版本中使用还是没有问题的，可能使用rest客户端兼容性更好做一些。

Elasticsearch Java Rest API 手册

Transport Client

不设置集群名称

```
// on startup

//此步骤添加IP，至少一个，如果设置了"client.transport.sniff"= true 一个就够了，因为添加了自动嗅探配置
TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
        .addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("host1"), 9300))
        .addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("host2"), 9300));

// on shutdown  关闭client

client.close();
```

设置集群名称

```
Settings settings = Settings.builder()
        .put("cluster.name", "myClusterName").build();  //设置ES实例的名称
TransportClient client = new PreBuiltTransportClient(settings);
 //自动嗅探整个集群的状态，把集群中其他ES节点的ip添加到本地的客户端列表中
//Add transport addresses and do something with the client...
```

增加自动嗅探配置

```
Settings settings = Settings.builder()
        .put("client.transport.sniff", true).build();
TransportClient client = new PreBuiltTransportClient(settings);
```

其他配置

```
client.transport.ignore_cluster_name  //设置 true ，忽略连接节点集群名验证
client.transport.ping_timeout       //ping一个节点的响应时间 默认5秒
client.transport.nodes_sampler_interval //sample/ping 节点的时间间隔，默认是5s
```

> 对于ES Client，有两种形式，一个是TransportClient，一个是NodeClient。两个的区别为： TransportClient作为一个外部访问者，通过HTTP去请求ES的集群，对于集群而言，它是一个外部因素。 NodeClient顾名思义，是作为ES集群的一个节点，它是ES中的一环，其他的节点对它是感知的，不像TransportClient那样，ES集群对它一无所知。NodeClient通信的性能会更好，但是因为是ES的一环，所以它出问题，也会给ES集群带来问题。NodeClient可以设置不作为数据节点，在elasticsearch.yml中设置，这样就不会在此节点上分配数据。

如果用ES的节点，大家仁者见仁智者见智，各按所需。

实例

```
Settings esSettings = Settings.builder()

    .put("cluster.name", clusterName) //设置ES实例的名称

    .put("client.transport.sniff", true) //自动嗅探整个集群的状态，把集群中其他ES节点的ip添加到本地的客户端列表中

    .build();

    client = new PreBuiltTransportClient(esSettings);//初始化client较老版本发生了变化，此方法有几个重载方法，初始化插件等。

    //此步骤添加IP，至少一个，其实一个就够了，因为添加了自动嗅探配置

    client.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName(ip), esPort));
```

XPackTransportClient

如果 `ElasticSearch` 服务安装了 `x-pack` 插件，需要 `PreBuiltXPackTransportClient` 实例才能访问

使用Maven管理项目，把下面代码增加到 `pom.xml` ;

> 一定要修改默认仓库地址为https://artifacts.elastic.co/maven ，因为这个库没有上传到Maven中央仓库

```
<project ...>

   <repositories>
      <!-- add the elasticsearch repo -->
      <repository>
         <id>elasticsearch-releases</id>
         <url>https://artifacts.elastic.co/maven</url>
         <releases>
            <enabled>true</enabled>
         </releases>
         <snapshots>
            <enabled>false</enabled>
         </snapshots>
      </repository>
      ...
   </repositories>
   ...

   <dependencies>
      <!-- add the x-pack jar as a dependency -->
      <dependency>
         <groupId>org.elasticsearch.client</groupId>
         <artifactId>x-pack-transport</artifactId>
         <version>5.6.3</version>
      </dependency>
      ...
   </dependencies>
   ...

 </project>
```

实例

```
Settings settings = Settings.builder().put("cluster.name", "xxx"
)
                    .put("xpack.security.transport.ssl.enabled",
 false)
                    .put("xpack.security.user", "xxx:xxx")
                    .put("client.transport.sniff", true).build()
;
try {
    client = new PreBuiltXPackTransportClient(settings)
            .addTransportAddress(new InetSocketTransportAddress(
InetAddress.getByName("xxx.xxx.xxx.xxx"), 9300))
            .addTransportAddress(new InetSocketTransportAddress(
InetAddress.getByName("xxx.xxx.xxx.xxx"), 9300));
} catch (UnknownHostException e) {
    e.printStackTrace();
}
```

更多请浏览 dayu-spring-boot-starter 开源项目

Document APIs

本节介绍以下 CRUD API：

单文档 APIs

- Index API
- Get API
- Delete API
- Delete By Query API
- Update API

多文档 APIs

- Multi Get API
- Bulk API
- Using Bulk Processor

Multi Get API Bulk API

> 注意:所有的单文档的CRUD API，index参数只能接受单一的索引库名称，或者是一个指向单一索引库的alias。

Index API

Index API 允许我们存储一个JSON格式的文档，使数据可以被搜索。文档通过index、type、id唯一确定。我们可以自己提供一个id，或者也使用Index API 为我们自动生成一个。

这里有几种不同的方式来产生JSON格式的文档(document)：

- 手动方式，使用原生的byte[]或者String
- 使用Map方式，会自动转换成与之等价的JSON
- 使用第三方库来序列化beans，如Jackson
- 使用内置的帮助类 XContentFactory.jsonBuilder()

手动方式

数据格式

```
String json = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
    "}";
```

实例

```
/**
 * 手动生成JSON
 */
@Test
public void CreateJSON(){

    String json = "{" +
            "\"user\":\"fendo\"," +
            "\"postDate\":\"2013-01-30\"," +
            "\"message\":\"Hell word\"" +
        "}";

    IndexResponse response = client.prepareIndex("fendo", "fendo
date")
            .setSource(json)
            .get();
    System.out.println(response.getResult());

}
```

Map方式

Map是key:value数据类型，可以代表json结构.

```
Map<String, Object> json = new HashMap<String, Object>();
json.put("user","kimchy");
json.put("postDate",new Date());
json.put("message","trying out Elasticsearch");
```

实例

```
 /**
 * 使用集合
 */
@Test
public void CreateList(){

    Map<String, Object> json = new HashMap<String, Object>();
    json.put("user","kimchy");
    json.put("postDate","2013-01-30");
    json.put("message","trying out Elasticsearch");

    IndexResponse response = client.prepareIndex("fendo", "fendo
date")
            .setSource(json)
            .get();
    System.out.println(response.getResult());

}
```

序列化方式

ElasticSearch已经使用了jackson，可以直接使用它把javabean转为json.

```
import com.fasterxml.jackson.databind.*;

// instance a json mapper
ObjectMapper mapper = new ObjectMapper(); // create once, reuse

// generate json
byte[] json = mapper.writeValueAsBytes(yourbeaninstance);
```

实例

```
/**
 * 使用JACKSON序列化
 * @throws Exception
 */
@Test
public void CreateJACKSON() throws Exception{

    CsdnBlog csdn=new CsdnBlog();
    csdn.setAuthor("fendo");
    csdn.setContent("这是JAVA书籍");
    csdn.setTag("C");
    csdn.setView("100");
    csdn.setTitile("编程");
    csdn.setDate(new Date().toString());

    // instance a json mapper
    ObjectMapper mapper = new ObjectMapper(); // create once, re
use

    // generate json
    byte[] json = mapper.writeValueAsBytes(csdn);

    IndexResponse response = client.prepareIndex("fendo", "fendo
date")
            .setSource(json)
            .get();
    System.out.println(response.getResult());
}
```

XContentBuilder帮助类方式

ElasticSearch提供了一个内置的帮助类XContentBuilder来产生JSON文档

```
// Index name
String _index = response.getIndex();
// Type name
String _type = response.getType();
// Document ID (generated or not)
String _id = response.getId();
// Version (if it's the first time you index this document, you
will get: 1)
long _version = response.getVersion();
// status has stored current instance statement.
RestStatus status = response.status();
```

实例

```
/**
 * 使用ElasticSearch 帮助类
 * @throws IOException
 */
@Test
public void CreateXContentBuilder() throws IOException{

    XContentBuilder builder = XContentFactory.jsonBuilder()
            .startObject()
                .field("user", "ccse")
                .field("postDate", new Date())
                .field("message", "this is Elasticsearch")
            .endObject();

    IndexResponse response = client.prepareIndex("fendo", "fendo
data").setSource(builder).get();
    System.out.println("创建成功!");


}
```

综合实例

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAdd
ress;
import org.elasticsearch.common.xcontent.XContentBuilder;
import org.elasticsearch.common.xcontent.XContentFactory;
import org.elasticsearch.transport.client.PreBuiltTransportClien
t;
import org.junit.Before;
import org.junit.Test;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class CreateIndex {

    private TransportClient client;

    @Before
    public void getClient() throws Exception{
        //设置集群名称
        Settings settings = Settings.builder().put("cluster.name
", "my-application").build();// 集群名
        //创建client
        client  = new PreBuiltTransportClient(settings)
                .addTransportAddress(new InetSocketTransportAddr
ess(InetAddress.getByName("127.0.0.1"), 9300));
    }

    /**
     * 手动生成JSON
     */
```

```
    @Test
    public void CreateJSON(){

        String json = "{" +
                "\"user\":\"fendo\"," +
                "\"postDate\":\"2013-01-30\"," +
                "\"message\":\"Hell word\"" +
            "}";

        IndexResponse response = client.prepareIndex("fendo", "f
endodate")
                .setSource(json)
                .get();
        System.out.println(response.getResult());

    }



    /**
     * 使用集合
     */
    @Test
    public void CreateList(){

        Map<String, Object> json = new HashMap<String, Object>()
;
        json.put("user","kimchy");
        json.put("postDate","2013-01-30");
        json.put("message","trying out Elasticsearch");

        IndexResponse response = client.prepareIndex("fendo", "f
endodate")
                .setSource(json)
                .get();
        System.out.println(response.getResult());

    }

    /**
     * 使用JACKSON序列化
```

```
     * @throws Exception
     */
    @Test
    public void CreateJACKSON() throws Exception{

        CsdnBlog csdn=new CsdnBlog();
        csdn.setAuthor("fendo");
        csdn.setContent("这是JAVA书籍");
        csdn.setTag("C");
        csdn.setView("100");
        csdn.setTitile("编程");
        csdn.setDate(new Date().toString());

        // instance a json mapper
        ObjectMapper mapper = new ObjectMapper(); // create once
, reuse

        // generate json
        byte[] json = mapper.writeValueAsBytes(csdn);

        IndexResponse response = client.prepareIndex("fendo", "f
endodate")
                .setSource(json)
                .get();
        System.out.println(response.getResult());
    }

    /**
     * 使用ElasticSearch 帮助类
     * @throws IOException
     */
    @Test
    public void CreateXContentBuilder() throws IOException{

        XContentBuilder builder = XContentFactory.jsonBuilder()

                .startObject()
                    .field("user", "ccse")
                    .field("postDate", new Date())
                    .field("message", "this is Elasticsearch")
```

```
                .endObject();

        IndexResponse response = client.prepareIndex("fendo", "f
endodata").setSource(builder).get();
        System.out.println("创建成功!");



    }

}
```

> 你还可以通过startArray(string)和endArray()方法添加数组。.field()方法可以接受多种对象类型。你可以给它传递数字、日期、甚至其他XContentBuilder对象。

Get API

根据id查看文档：

```
GetResponse response = client.prepareGet("twitter", "tweet", "1"
).get();
```

更多请查看 rest get API 文档

配置线程

`operationThreaded` 设置为 `true` 是在不同的线程里执行此次操作

下面的例子是`operationThreaded` 设置为 `false` ：

```
GetResponse response = client.prepareGet("twitter", "tweet", "1"
)
        .setOperationThreaded(false)
        .get();
```

Delete API

根据ID删除：

```
DeleteResponse response = client.prepareDelete("twitter", "tweet
", "1").get();
```

更多请查看 delete API 文档

配置线程

`operationThreaded` 设置为 `true` 是在不同的线程里执行此次操作

下面的例子是`operationThreaded` 设置为 `false` ：

```
GetResponse response = client.prepareGet("twitter", "tweet", "1"
)
        .setOperationThreaded(false)
        .get();
```

```
DeleteResponse response = client.prepareDelete("twitter", "tweet
", "1")
        .setOperationThreaded(false)
        .get();
```

Delete By Query API

通过查询条件删除

```
BulkByScrollResponse response =
    DeleteByQueryAction.INSTANCE.newRequestBuilder(client)
        .filter(QueryBuilders.matchQuery("gender", "male")) //查
询条件
        .source("persons") //index(索引名)
        .get();  //执行

long deleted = response.getDeleted(); //删除文档的数量
```

如果需要执行的时间比较长，可以使用异步的方式处理,结果在回调里面获取

```
DeleteByQueryAction.INSTANCE.newRequestBuilder(client)
    .filter(QueryBuilders.matchQuery("gender", "male"))      //
查询
    .source("persons")                //index(索引名)

    .execute(new ActionListener<BulkByScrollResponse>() {     //
回调监听
        @Override
        public void onResponse(BulkByScrollResponse response) {
            long deleted = response.getDeleted();   //删除文档的数
量
        }
        @Override
        public void onFailure(Exception e) {
            // Handle the exception
        }
    });
```

Update API

有两种方式更新索引：

- 创建 `UpdateRequest` ,通过client发送；
- 使用 `prepareUpdate()` 方法；

使用UpdateRequest

```
UpdateRequest updateRequest = new UpdateRequest();
updateRequest.index("index");
updateRequest.type("type");
updateRequest.id("1");
updateRequest.doc(jsonBuilder()
        .startObject()
            .field("gender", "male")
        .endObject());
client.update(updateRequest).get();
```

使用 `prepareUpdate()` 方法

> 这里官方的示例有问题，new Script（）参数错误，所以一下代码是我自己写的（2017/11/10）

```
client.prepareUpdate("ttl", "doc", "1")
        .setScript(new Script("ctx._source.gender = \"male\""  , ScriptService.ScriptType.INLINE, null, null))//脚本可以是本地文件存储的，如果使用文件存储的脚本，需要设置 ScriptService.ScriptType.FILE
        .get();

client.prepareUpdate("ttl", "doc", "1")
        .setDoc(jsonBuilder()   //合并到现有文档
            .startObject()
                .field("gender", "male")
            .endObject())
        .get();
```

Update by script

使用脚本更新文档

```
UpdateRequest updateRequest = new UpdateRequest("ttl", "doc", "1
")
        .script(new Script("ctx._source.gender = \"male\""));
client.update(updateRequest).get();
```

Update by merging documents

合并文档

```
UpdateRequest updateRequest = new UpdateRequest("index", "type",
 "1")
        .doc(jsonBuilder()
            .startObject()
                .field("gender", "male")
            .endObject());
client.update(updateRequest).get();
```

Upsert

更新插入,如果存在文档就更新，如果不存在就插入

```
IndexRequest indexRequest = new IndexRequest("index", "type", "1
")
        .source(jsonBuilder()
            .startObject()
                .field("name", "Joe Smith")
                .field("gender", "male")
            .endObject());
UpdateRequest updateRequest = new UpdateRequest("index", "type",
 "1")
        .doc(jsonBuilder()
            .startObject()
                .field("gender", "male")
            .endObject())
        .upsert(indexRequest); //如果不存在此文档 ，就增加 `indexReq
uest`
client.update(updateRequest).get();
```

如果 `index/type/1` 存在，类似下面的文档：

```
{
    "name"  : "Joe Dalton",
    "gender": "male"
}
```

如果不存在，会插入新的文档：

```
{
    "name" : "Joe Smith",
    "gender": "male"
}
```

Multi Get API

一次获取多个文档

```
MultiGetResponse multiGetItemResponses = client.prepareMultiGet(
)
    .add("twitter", "tweet", "1") //一个id的方式
    .add("twitter", "tweet", "2", "3", "4") //多个id的方式
    .add("another", "type", "foo")  //可以从另外一个索引获取
    .get();

for (MultiGetItemResponse itemResponse : multiGetItemResponses)
{ //迭代返回值
    GetResponse response = itemResponse.getResponse();
    if (response.isExists()) {      //判断是否存在
        String json = response.getSourceAsString(); //_source 字
段
    }
}
```

更多请浏览REST multi get 文档

Bulk API

Bulk API，批量插入：

```
import static org.elasticsearch.common.xcontent.XContentFactory.
*;
```

```
BulkRequestBuilder bulkRequest = client.prepareBulk();

// either use client#prepare, or use Requests# to directly build
 index/delete requests
bulkRequest.add(client.prepareIndex("twitter", "tweet", "1")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "trying out Elasticsea
rch")
                    .endObject()
                  )
        );

bulkRequest.add(client.prepareIndex("twitter", "tweet", "2")
        .setSource(jsonBuilder()
                    .startObject()
                        .field("user", "kimchy")
                        .field("postDate", new Date())
                        .field("message", "another post")
                    .endObject()
                  )
        );

BulkResponse bulkResponse = bulkRequest.get();
if (bulkResponse.hasFailures()) {
    // process failures by iterating through each bulk response
item
    //处理失败
}
```

使用 Bulk Processor

BulkProcessor 提供了一个简单的接口，在给定的大小数量上定时批量自动请求

创建 `BulkProcessor` 实例

首先创建 `BulkProcessor` 实例

```
import org.elasticsearch.action.bulk.BackoffPolicy;
import org.elasticsearch.action.bulk.BulkProcessor;
import org.elasticsearch.common.unit.ByteSizeUnit;
import org.elasticsearch.common.unit.ByteSizeValue;
import org.elasticsearch.common.unit.TimeValue;
```

```
BulkProcessor bulkProcessor = BulkProcessor.builder(
        client,  //增加elasticsearch客户端
        new BulkProcessor.Listener() {
            @Override
            public void beforeBulk(long executionId,
                                   BulkRequest request) { ... }
//调用bulk之前执行 ，例如你可以通过request.numberOfActions()方法知道nu
mberOfActions

            @Override
            public void afterBulk(long executionId,
                                  BulkRequest request,
                                  BulkResponse response) { ... }
 //调用bulk之后执行 ，例如你可以通过request.hasFailures()方法知道是否执
行失败

            @Override
            public void afterBulk(long executionId,
                                  BulkRequest request,
                                  Throwable failure) { ... } //
调用失败抛 Throwable
        })
        .setBulkActions(10000) //每次10000请求
        .setBulkSize(new ByteSizeValue(5, ByteSizeUnit.MB)) //拆
成5mb一块
        .setFlushInterval(TimeValue.timeValueSeconds(5)) //无论请
求数量多少，每5秒钟请求一次。
        .setConcurrentRequests(1) //设置并发请求的数量。值为0意味着只
允许执行一个请求。值为1意味着允许1并发请求。
        .setBackoffPolicy(
            BackoffPolicy.exponentialBackoff(TimeValue.timeValue
Millis(100), 3))//设置自定义重复请求机制，最开始等待100毫秒，之后成倍更加
，重试3次，当一次或多次重复请求失败后因为计算资源不够抛出 EsRejectedExecu
tionException 异常，可以通过BackoffPolicy.noBackoff()方法关闭重试机制
        .build();
```

BulkProcessor 默认设置

- bulkActions 1000

- bulkSize 5mb
- 不设置flushInterval
- concurrentRequests 为 1 ，异步执行
- backoffPolicy 重试 8次，等待50毫秒

增加requests

然后增加 `requests` 到 `BulkProcessor`

```
bulkProcessor.add(new IndexRequest("twitter", "tweet", "1").sour
ce(/* your doc here */));
bulkProcessor.add(new DeleteRequest("twitter", "tweet", "2"));
```

关闭 Bulk Processor

当所有文档都处理完成，使用 `awaitClose` 或 `close` 方法关闭 `BulkProcessor` :

```
bulkProcessor.awaitClose(10, TimeUnit.MINUTES);
```

或

```
bulkProcessor.close();
```

在测试中使用Bulk Processor

如果你在测试种使用 `Bulk Processor` 可以执行同步方法

```
BulkProcessor bulkProcessor = BulkProcessor.builder(client, new
BulkProcessor.Listener() { /* Listener methods */ })
        .setBulkActions(10000)
        .setConcurrentRequests(0)
        .build();

// Add your requests
bulkProcessor.add(/* Your requests */);

// Flush any remaining requests
bulkProcessor.flush();

// Or close the bulkProcessor if you don't need it anymore
bulkProcessor.close();

// Refresh your indices
client.admin().indices().prepareRefresh().get();

// Now you can start searching!
client.prepareSearch().get();
```

搜索API

搜索查询，返回查询匹配的结果，搜索一个index / type 或者多个index / type，可以使用 query Java API 作为查询条件，下面是例子：

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.action.search.SearchType;
import org.elasticsearch.index.query.QueryBuilders.*;
```

```
SearchResponse response = client.prepareSearch("index1", "index2
")
        .setTypes("type1", "type2")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.termQuery("multi", "test"))
            // Query 查询条件
        .setPostFilter(QueryBuilders.rangeQuery("age").from(12).
to(18))     // Filter 过滤
        .setFrom(0).setSize(60).setExplain(true)
        .get();
```

所有的参数都是可选的，下面是最简单的调用：

```
// MatchAll on the whole cluster with all default options
SearchResponse response = client.prepareSearch().get();
```

> 尽管Java API默认提供 `QUERY_AND_FETCH` 和 `DFS_QUERY_AND_FETCH` 两种 search types ，但是这种模式应该由系统选择，用户不要手动指定

更多请移步 REST search 文档

Using scrolls in Java

首先需要阅读 scroll documentation

> 一般搜索请求都是返回一"页"数据，无论数据量多大都一起返回给用户，Scroll API可以允许我们检索大量数据（甚至全部数据）。Scroll API允许我们做一个初始阶段搜索并且持续批量从Elasticsearch里拉取结果直到没有结果剩下。这有点像传统数据库里的cursors（游标）。 Scroll API的创建并不是为了实时的用户响应，而是为了处理大量的数据（Scrolling is not intended for real time user requests, but rather for processing large amounts of data）。从 scroll 请求返回的结果只是反映了 search 发生那一时刻的索引状态，就像一个快照 (The results that are returned from a scroll request reflect the state of the index at the time that the initial search request was made, like a snapshot in time)。后续的对文档的改动（索引、更新或者删除）都只会影响后面的搜索请求。

```
import static org.elasticsearch.index.query.QueryBuilders.*;
```

```
QueryBuilder qb = termQuery("multi", "test");

SearchResponse scrollResp = client.prepareSearch(test)
        .addSort(FieldSortBuilder.DOC_FIELD_NAME, SortOrder.ASC)
        .setScroll(new TimeValue(60000)) //为了使用 scroll，初始搜
索请求应该在查询中指定 scroll 参数，告诉 Elasticsearch 需要保持搜索的上
下文环境多长时间（滚动时间）
        .setQuery(qb)
        .setSize(100).get(); //max of 100 hits will be returned
for each scroll
//Scroll until no hits are returned
do {
    for (SearchHit hit : scrollResp.getHits().getHits()) {
        //Handle the hit...
    }

    scrollResp = client.prepareSearchScroll(scrollResp.getScroll
Id()).setScroll(new TimeValue(60000)).execute().actionGet();
} while(scrollResp.getHits().getHits().length != 0); // Zero hit
s mark the end of the scroll and the while loop.
```

> 如果超过滚动时间，继续使用该滚动ID搜索数据，则会报错：

```
Caused by: SearchContextMissingException[No search context found
 for id [2861]]
    at org.elasticsearch.search.SearchService.findContext(Search
Service.java:613)
    at org.elasticsearch.search.SearchService.executeQueryPhase(
SearchService.java:403)
    at org.elasticsearch.search.action.SearchServiceTransportAct
ion$SearchQueryScrollTransportHandler.messageReceived(SearchServ
iceTransportAction.java:384)
    at org.elasticsearch.search.action.SearchServiceTransportAct
ion$SearchQueryScrollTransportHandler.messageReceived(SearchServ
iceTransportAction.java:381)
    at org.elasticsearch.transport.TransportRequestHandler.messa
geReceived(TransportRequestHandler.java:33)
    at org.elasticsearch.transport.RequestHandlerRegistry.proces
sMessageReceived(RequestHandlerRegistry.java:75)
    at org.elasticsearch.transport.TransportService$4.doRun(Tran
sportService.java:376)
    at org.elasticsearch.common.util.concurrent.AbstractRunnable
.run(AbstractRunnable.java:37)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadP
oolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(Thread
PoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
```

> 虽然当滚动有效时间已过，搜索上下文(Search Context)会自动被清除，但是一值保持滚动代价也是很大的，所以当我们不在使用滚动时要尽快使用Clear-Scroll API进行清除。

清除Scroll

```
  /**
     * 清除滚动ID
     * @param client
     * @param scrollIdList
     * @return
     */
    public static boolean clearScroll(Client client, List<String
> scrollIdList){
        ClearScrollRequestBuilder clearScrollRequestBuilder = cl
ient.prepareClearScroll();
        clearScrollRequestBuilder.setScrollIds(scrollIdList);
        ClearScrollResponse response = clearScrollRequestBuilder
.get();
        return response.isSucceeded();
    }
    /**
     * 清除滚动ID
     * @param client
     * @param scrollId
     * @return
     */
    public static boolean clearScroll(Client client, String scro
llId){
        ClearScrollRequestBuilder clearScrollRequestBuilder = cl
ient.prepareClearScroll();
        clearScrollRequestBuilder.addScrollId(scrollId);
        ClearScrollResponse response = clearScrollRequestBuilder
.get();
        return response.isSucceeded();
    }
```

实例

```

public class ScrollsAPI extends ElasticsearchClientBase {

    private String scrollId;

```

```
    @Test
    public void testScrolls() throws Exception {

        SearchResponse scrollResp = client.prepareSearch("twitte
r")
                .addSort(FieldSortBuilder.DOC_FIELD_NAME, SortOr
der.ASC)
                .setScroll(new TimeValue(60000)) //为了使用 scrol
l，初始搜索请求应该在查询中指定 scroll 参数，告诉 Elasticsearch 需要保持
搜索的上下文环境多长时间（滚动时间）
                .setQuery(QueryBuilders.termQuery("user", "kimch
y"))                 // Query 查询条件
                .setSize(5).get(); //max of 100 hits will be ret
urned for each scroll
        //Scroll until no hits are returned

        scrollId = scrollResp.getScrollId();
        do {
            for (SearchHit hit : scrollResp.getHits().getHits())
 {
                //Handle the hit...

                System.out.println("" + hit.getSource().toString
());
            }

            scrollResp = client.prepareSearchScroll(scrollId).se
tScroll(new TimeValue(60000)).execute().actionGet();
        }
        while (scrollResp.getHits().getHits().length != 0); // Z
ero hits mark the end of the scroll and the while loop.
    }

    @Override
    public void tearDown() throws Exception {
        ClearScrollRequestBuilder clearScrollRequestBuilder = cl
ient.prepareClearScroll();
        clearScrollRequestBuilder.addScrollId(scrollId);
        ClearScrollResponse response = clearScrollRequestBuilder
.get();
```

```
        if (response.isSucceeded()) {
            System.out.println("成功清除");
        }

        super.tearDown();
    }
}
```

- ScrollsAPI.java
- 本手册完整实例

MultiSearch API

multi search API 允许在同一API中执行多个搜索请求。它的端点（endpoint）是 _msearch 。

首先请看MultiSearch API Query 文档

```
SearchRequestBuilder srb1 = client
    .prepareSearch().setQuery(QueryBuilders.queryStringQuery("el
asticsearch")).setSize(1);
SearchRequestBuilder srb2 = client
    .prepareSearch().setQuery(QueryBuilders.matchQuery("name", "
kimchy")).setSize(1);

MultiSearchResponse sr = client.prepareMultiSearch()
        .add(srb1)
        .add(srb2)
        .get();

// You will get all individual responses from MultiSearchRespons
e#getResponses()
long nbHits = 0;
for (MultiSearchResponse.Item item : sr.getResponses()) {
    SearchResponse response = item.getResponse();
    nbHits += response.getHits().getTotalHits();
}
```

实例

- MultiSearchAPI.java
- 本手册完整实例

Using Aggregations

下面的代码演示了如何在搜索中添加两个聚合：

> 聚合框架有助于根据搜索查询提供聚合数据。它是基于简单的构建块也称为整合，整合就是将复杂的数据摘要有序的放在一块。
>
> 聚合可以被看做是从一组文件中获取分析信息的一系列工作的统称。聚合的实现过程就是定义这个文档集的过程（例如，在搜索请求的基础上，执行查询/过滤，才能得到高水平的聚合结果）。

```
SearchResponse sr = client.prepareSearch()
    .setQuery(QueryBuilders.matchAllQuery())
    .addAggregation(
            AggregationBuilders.terms("agg1").field("field")
    )
    .addAggregation(
            AggregationBuilders.dateHistogram("agg2")
                    .field("birth")
                    .dateHistogramInterval(DateHistogramInterval
.YEAR)
    )
    .get();

// Get your facet results
Terms agg1 = sr.getAggregations().get("agg1");
Histogram agg2 = sr.getAggregations().get("agg2");
```

详细文档请看 Aggregations Java API

Terminate After

获取文档的最大数量，如果设置了，需要通过 `SearchResponse` 对象里的 `isTerminatedEarly()` 判断返回文档是否达到设置的数量：

```
SearchResponse sr = client.prepareSearch(INDEX)
    .setTerminateAfter(1000)    //如果达到这个数量，提前终止
    .get();

if (sr.isTerminatedEarly()) {
    // We finished early
}
```

Search Template

首先查看 Search Template 文档

> /_search/template endpoint 允许我们在执行搜索请求和使用模板参数填充现有模板之前，能够使用 `mustache` 语言预先呈现搜索请求。

将模板参数定义为 `Map <String，Object>` ：

```
Map<String, Object> template_params = new HashMap<>();
template_params.put("param_gender", "male");
```

可以在 `config/scripts` 中使用存储的 `search templates` 。例如，有一个名为 `config/scripts/template_gender.mustache` 的文件，其中包含：

```
{
    "query" : {
        "match" : {
            "gender" : ""
        }
    }
}
```

创建 `search templates` 请求：

```
SearchResponse sr = new SearchTemplateRequestBuilder(client)
    .setScript("template_gender")                 //template 名
    .setScriptType(ScriptService.ScriptType.FILE) //template 存储在 gender_template.mustache 磁盘上
    .setScriptParams(template_params)             //参数
    .setRequest(new SearchRequest())              //设置执行的context（ie: 这里定义索引名称）
    .get()
    .getResponse();
```

还可以将 `template` 存储在 `cluster state` 中:

> cluster state是全局性信息，包含了整个群集中所有分片的元信息(规则，位置，大小等信息)，并保持每个每节的信息同步。 参考：《为什么ElasticSearch应用开发者需要了解cluster state》

```
client.admin().cluster().preparePutStoredScript()
    .setScriptLang("mustache")
    .setId("template_gender")
    .setSource(new BytesArray(
        "{\n" +
        "    \"query\" : {\n" +
        "        \"match\" : {\n" +
        "            \"gender\" : \"\"\n" +
        "        }\n" +
        "    }\n" +
        "}")).get();
```

使用 `ScriptService.ScriptType.STORED` 执行一个存储的 `templates`：

```
SearchResponse sr = new SearchTemplateRequestBuilder(client)
        .setScript("template_gender")          //template 名
        .setScriptType(ScriptType.STORED)      //template 存储在
cluster state 上
        .setScriptParams(template_params)      //参数
        .setRequest(new SearchRequest())       //设置执行的context
（ie: 这里定义索引名称）
        .get()                                 //执行获取template
 请求
        .getResponse();
```

也可以执行 内联(`inline`) `templates` ：

```
sr = new SearchTemplateRequestBuilder(client)
        .setScript("{\n" +                            //template
 名
                "        \"query\" : {\n" +
                "            \"match\" : {\n" +
                "                \"gender\" : \"\"\n" +
                "            }\n" +
                "        }\n" +
                "}")
        .setScriptType(ScriptType.INLINE)    //template 是内联传递
的
        .setScriptParams(template_params)    //参数
        .setRequest(new SearchRequest())     //设置执行的context（
ie: 这里定义索引名称）
        .get()                               //执行获取template 请
求
        .getResponse();
```

Aggregations

聚合

Elasticsearch提供完整的Java API来使用聚合。 请参阅聚合指南。

使用 `AggregationBuilders` 构建对象，增加到搜索请求中:

```
import org.elasticsearch.search.aggregations.AggregationBuilders
;
```

```
SearchResponse sr = node.client().prepareSearch()
        .setQuery( /* your query */ )
        .addAggregation( /* add an aggregation */ )
        .execute().actionGet();
```

Structuring aggregations

结构化聚合

如 Aggregations guide 中所述，可以在聚合中定义子聚合。

聚合可能是 `Metrics` 聚合(一个跟踪和计算指标的聚合)或者 `Bucket` 聚合 (构建桶聚合)

例如，这里是一个3级聚合组成的聚合：

- Terms aggregation (bucket)
- Date Histogram aggregation (bucket)
- Average aggregation (metric)

```
SearchResponse sr = node.client().prepareSearch()
    .addAggregation(
        AggregationBuilders.terms("by_country").field("country")
        .subAggregation(AggregationBuilders.dateHistogram("by_ye
ar")
            .field("dateOfBirth")
            .dateHistogramInterval(DateHistogramInterval.YEAR)
            .subAggregation(AggregationBuilders.avg("avg_childre
n").field("children"))
        )
    )
    .execute().actionGet();
```

Metrics aggregations

> 计算度量这类的聚合操作是以使用一种方式或者从文档中提取需要聚合的值为基础的。这些数据不但可以从文档（使用数据属性）的属性中提取出来，也可以使用脚本生成。
>
> 数值计量聚合操作是能够产生具体的数值的一种计量聚合操作。一些聚合操作输出单个的计量数值（例如avg），并且被称作single-value numeric metric aggregation，其他产生多个计量数值（例如 stats）的称作 multi-value numeric metrics aggregation。这两种不同的聚合操作只有在桶聚合的子聚合操作中才会有不同的表现（有些桶聚合可以基于每个的数值计量来对返回的桶进行排序）。

Min Aggregatione 最小值聚合

下面是如何用Java API 使用最小值聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
MinAggregationBuilder aggregation =
        AggregationBuilders
                .min("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.min.Min;
```

```
// sr is here your SearchResponse object
Min agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

Max Aggregation 最大值聚合

下面是如何用Java API 使用最大值聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
MaxAggregationBuilder aggregation =
        AggregationBuilders
                .max("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.max.Max;
```

```
// sr is here your SearchResponse object
Max agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

Sum Aggregation 求和聚合

下面是如何用Java API 使用求和聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
SumAggregationBuilder aggregation =
        AggregationBuilders
                .sum("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.sum.Sum;
```

```
// sr is here your SearchResponse object
Sum agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

Avg Aggregation 平均值聚合

下面是如何用Java API 使用平均值聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AvgAggregationBuilder aggregation =
        AggregationBuilders
                .avg("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.avg.Avg;
```

```
// sr is here your SearchResponse object
Avg agg = sr.getAggregations().get("agg");
double value = agg.getValue();
```

Stats Aggregation 统计聚合

> 统计聚合——基于文档的某个值，计算出一些统计信息（min、max、sum、count、avg），用于计算的值可以是特定的数值型字段，也可以通过脚本计算而来。

下面是如何用Java API 使用统计聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
StatsAggregationBuilder aggregation =
        AggregationBuilders
                .stats("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.stats.Stats
;
```

```
// sr is here your SearchResponse object
Stats agg = sr.getAggregations().get("agg");
double min = agg.getMin();
double max = agg.getMax();
double avg = agg.getAvg();
double sum = agg.getSum();
long count = agg.getCount();
```

Extended Stats Aggregation 扩展统计聚合

> 扩展统计聚合——基于文档的某个值，计算出一些统计信息（比普通的stats聚合多了sum_of_squares、variance、std_deviation、std_deviation_bounds），用于计算的值可以是特定的数值型字段，也可以通过脚本计算而来。

下面是如何用Java API 使用扩展统计聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
ExtendedStatsAggregationBuilder aggregation =
        AggregationBuilders
                .extendedStats("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.stats.exten
ded.ExtendedStats;
```

```
// sr is here your SearchResponse object
ExtendedStats agg = sr.getAggregations().get("agg");
double min = agg.getMin();
double max = agg.getMax();
double avg = agg.getAvg();
double sum = agg.getSum();
long count = agg.getCount();
double stdDeviation = agg.getStdDeviation();
double sumOfSquares = agg.getSumOfSquares();
double variance = agg.getVariance();
```

Value Count Aggregation 值计数聚合

> 值计数聚合——计算聚合文档中某个值的个数, 用于计算的值可以是特定的数值型字段，也可以通过脚本计算而来。
>
> 该聚合一般域其它 single-value 聚合联合使用，比如在计算一个字段的平均值的时候，可能还会关注这个平均值是由多少个值计算而来。

下面是如何用Java API 使用值计数聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
ValueCountAggregationBuilder aggregation =
        AggregationBuilders
                .count("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.valuecount.
ValueCount;
```

```
// sr is here your SearchResponse object
ValueCount agg = sr.getAggregations().get("agg");
long value = agg.getValue();
```

Percentile Aggregation 百分百聚合

> 百分百聚合——基于聚合文档中某个数值类型的值，求这些值中的一个或者多个百分比, 用于计算的值可以是特定的数值型字段，也可以通过脚本计算而来。

下面是如何用Java API 使用百分百聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
PercentilesAggregationBuilder aggregation =
        AggregationBuilders
                .percentiles("agg")
                .field("height");
```

可以提供百分位数，而不是使用默认值：

```
PercentilesAggregationBuilder aggregation =
        AggregationBuilders
                .percentiles("agg")
                .field("height")
                .percentiles(1.0, 5.0, 10.0, 20.0, 30.0, 75.0, 9
5.0, 99.0);
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.percentiles
.Percentile;
import org.elasticsearch.search.aggregations.metrics.percentiles
.Percentiles;
```

```
// sr is here your SearchResponse object
Percentiles agg = sr.getAggregations().get("agg");
// For each entry
for (Percentile entry : agg) {
    double percent = entry.getPercent();    // Percent
    double value = entry.getValue();        // Value

    logger.info("percent [{}], value [{}]", percent, value);
}
```

大概输出：

```
percent [1.0], value [0.814338896154595]
percent [5.0], value [0.8761912455821302]
percent [25.0], value [1.173346540141847]
percent [50.0], value [1.5432023318692198]
percent [75.0], value [1.923915462033674]
percent [95.0], value [2.2273644908535335]
percent [99.0], value [2.284989339108279]
```

Percentile Ranks Aggregation 百分比等级聚合

> 一个multi-value指标聚合，它通过从聚合文档中提取数值来计算一个或多个百分比。这些值可以从特定数值字段中提取，也可以由提供的脚本生成。
>
> 注意：请参考百分比（通常）近视值（percentiles are (usually approximate)）和压缩（Compression）以获得关于近视值的建议和内存使用的百分比排位聚合。百分比排位展示那些在某一值之下的观测值的百分比。例如，假如某一直大于等于被观测值的95%，则称其为第95百分位数。假设你的数据由网页加载时间组成。你可能有一个服务协议，95%页面需要在15ms加载完全，99%页面在30ms加载完全。

下面是如何用Java API 使用百分比等级聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
PercentilesAggregationBuilder aggregation =
        AggregationBuilders
                .percentiles("agg")
                .field("height");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.percentiles
.Percentile;
import org.elasticsearch.search.aggregations.metrics.percentiles
.PercentileRanks;
```

```
// sr is here your SearchResponse object
PercentileRanks agg = sr.getAggregations().get("agg");
// For each entry
for (Percentile entry : agg) {
    double percent = entry.getPercent();    // Percent
    double value = entry.getValue();        // Value

    logger.info("percent [{}], value [{}]", percent, value);
}
```

大概输出：

```
percent [29.664353095090945], value [1.24]
percent [73.9335313461868], value [1.91]
percent [94.40095147327283], value [2.22]
```

Cardinality Aggregation 基数聚合

> 基于文档的某个值，计算文档非重复的个数（去重计数）。。这些值可以从特定数值字段中提取，也可以由提供的脚本生成。

下面是如何用Java API 使用基数聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
CardinalityAggregationBuilder aggregation =
        AggregationBuilders
                .cardinality("agg")
                .field("tags");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.cardinality
.Cardinality;
```

```
// sr is here your SearchResponse object
Cardinality agg = sr.getAggregations().get("agg");
long value = agg.getValue();
```

Geo Bounds Aggregation 地理边界聚合

> 地理边界聚合——基于文档的某个字段（geo-point类型字段），计算出该字段所有地理坐标点的边界（左上角/右下角坐标点）。

下面是如何用Java API 使用地理边界聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
GeoBoundsBuilder aggregation =
        GeoBoundsAggregationBuilder
                .geoBounds("agg")
                .field("address.location")
                .wrapLongitude(true);
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.metrics.geobounds.G
eoBounds;
```

```
// sr is here your SearchResponse object
GeoBounds agg = sr.getAggregations().get("agg");
GeoPoint bottomRight = agg.bottomRight();
GeoPoint topLeft = agg.topLeft();
logger.info("bottomRight {}, topLeft {}", bottomRight, topLeft);
```

大概会输出：

```
bottomRight [40.70500764381921, 13.952946866893775], topLeft [53
.49603022435221, -4.190029308156676]
```

Top Hits Aggregation 最高匹配权值聚合

> 最高匹配权值聚合——跟踪聚合中相关性最高的文档。该聚合一般用做 sub-aggregation，以此来聚合每个桶中的最高匹配的文档。

下面是如何用Java API 使用最高匹配权值聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
    AggregationBuilders
        .terms("agg").field("gender")
        .subAggregation(
            AggregationBuilders.topHits("top")
        );
```

大多数标准的搜索选项可以使用，比如：`from`，`size`，`sort`，`highlight`，`explain` …

```
AggregationBuilder aggregation =
    AggregationBuilders
        .terms("agg").field("gender")
        .subAggregation(
            AggregationBuilders.topHits("top")
                .explain(true)
                .size(1)
                .from(10)
        );
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
import org.elasticsearch.search.aggregations.metrics.tophits.Top
Hits;
```

```
// sr is here your SearchResponse object
Terms agg = sr.getAggregations().get("agg");

// For each entry
for (Terms.Bucket entry : agg.getBuckets()) {
    String key = entry.getKey();                    // bucket ke
y
    long docCount = entry.getDocCount();            // Doc count
    logger.info("key [{}], doc_count [{}]", key, docCount);

    // We ask for top_hits for each bucket
    TopHits topHits = entry.getAggregations().get("top");
    for (SearchHit hit : topHits.getHits().getHits()) {
        logger.info(" -> id [{}], _source [{}]", hit.getId(), hi
t.getSourceAsString());
    }
}
```

大概会输出：

```
key [male], doc_count [5107]
 -> id [AUnzSZze9k7PKXtq04x2], _source [{"gender":"male",...}]
 -> id [AUnzSZzj9k7PKXtq04x4], _source [{"gender":"male",...}]
 -> id [AUnzSZzl9k7PKXtq04x5], _source [{"gender":"male",...}]
key [female], doc_count [4893]
 -> id [AUnzSZzM9k7PKXtq04xy], _source [{"gender":"female",...}]
 -> id [AUnzSZzp9k7PKXtq04x8], _source [{"gender":"female",...}]
 -> id [AUnzSZ0W9k7PKXtq04yS], _source [{"gender":"female",...}]
```

Scripted Metric Aggregation

> 此功能为实验性的，不建议生产使用，所以也不做过多说明 有兴趣可以自己参考 官方文档

Bucket aggregations 桶分聚合

> Bucket aggregations 不像 metrics aggregations 那样计算指标，恰恰相反，它创建文档的buckets，每个buckets与标准（取决于聚合类型）相关联，它决定了当前上下文中的文档是否会“falls”到它。换句话说，bucket可以有效地定义文档集合。除了buckets本身，bucket集合还计算并返回“落入”每个bucket的文档数量。
>
> 与度量聚合相比，Bucket聚合可以保存子聚合，这些子聚合将针对由其“父”bucket聚合创建的bucket进行聚合。
>
> 有不同的bucket聚合器，每个具有不同的“bucketing”策略,一些定义一个单独的bucket，一些定义多个bucket的固定数量，另一些定义在聚合过程中动态创建bucket

Global Aggregation 全局聚合

> 定义搜索执行上下文中的所有文档的单个bucket，这个上下文由索引和您正在搜索的文档类型定义，但不受搜索查询本身的影响。
>
> 全局聚合器只能作为顶层聚合器放置，因为将全局聚合器嵌入到另一个分组聚合器中是没有意义的。

下面是如何使用 `Java API` 使用全局聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilders
    .global("agg")
    .subAggregation(AggregationBuilders.terms("genders").field("
gender"));
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.global.Globa
l;
```

```
// sr is here your SearchResponse object
Global agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

Filter Aggregation 过滤聚合

> 过滤聚合——基于一个条件，来对当前的文档进行过滤的聚合。

下面是如何使用 `Java API` 使用过滤聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilders
    .filter("agg", QueryBuilders.termQuery("gender", "male"));
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.filter.Filte
r;
```

```
// sr is here your SearchResponse object
Filter agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

Filters Aggregation 多过滤聚合

> 多过滤聚合——基于多个过滤条件，来对当前文档进行【过滤】的聚合，每个过滤都包含所有满足它的文档（多个bucket中可能重复）。

下面是如何使用 `Java API` 使用多过滤聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
    AggregationBuilders
        .filters("agg",
            new FiltersAggregator.KeyedFilter("men", QueryBuilde
rs.termQuery("gender", "male")),
            new FiltersAggregator.KeyedFilter("women", QueryBuil
ders.termQuery("gender", "female")));
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.filters.Filt
ers;
```

```
// sr is here your SearchResponse object
Filters agg = sr.getAggregations().get("agg");

// For each entry
for (Filters.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();            // bucket ke
y
    long docCount = entry.getDocCount();            // Doc count
    logger.info("key [{}], doc_count [{}]", key, docCount);
}
```

大概输出

```
key [men], doc_count [4982]
key [women], doc_count [5018]
```

Missing Aggregation 基于字段数据的单桶聚合

> 基于字段数据的单桶聚合，创建当前文档集上下文中缺少字段值的所有文档的bucket（桶）（有效地，丢失了一个字段或配置了NULL值集），此聚合器通常与其他字段数据桶聚合器（例如范围）结合使用，以返回由于缺少字段数据值而无法放置在任何其他存储区中的所有文档的信息。

下面是如何使用 `Java API` 使用基于字段数据的单桶聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilders.missing("agg").field("gender");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.missing.Missing;
```

```
// sr is here your SearchResponse object
Missing agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

Nested Aggregation 嵌套类型聚合

> 基于嵌套（nested）数据类型，把该【嵌套类型的信息】聚合到单个桶里，然后就可以对嵌套类型做进一步的聚合操作。

下面是如何使用 `Java API` 使用嵌套类型聚合

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilders
    .nested("agg", "resellers");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.nested.Neste
d;
```

```
// sr is here your SearchResponse object
Nested agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

Reverse nested Aggregation

> 一个特殊的单桶聚合，可以从嵌套文档中聚合父文档。实际上，这种聚合可以从嵌套的块结构中跳出来，并链接到其他嵌套的结构或根文档.这允许嵌套不是嵌套对象的一部分的其他聚合在嵌套聚合中。 reverse_nested 聚合必须定义在nested之中

下面是如何使用 `Java API` 使用Reverse nested Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
    AggregationBuilders
        .nested("agg", "resellers")
        .subAggregation(
                AggregationBuilders
                        .terms("name").field("resellers.name")
                        .subAggregation(
                                AggregationBuilders
                                        .reverseNested("reseller
_to_product")
                        )
        );
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.nested.Neste
d;
import org.elasticsearch.search.aggregations.bucket.nested.Rever
seNested;
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
```

```
// sr is here your SearchResponse object
Nested agg = sr.getAggregations().get("agg");
Terms name = agg.getAggregations().get("name");
for (Terms.Bucket bucket : name.getBuckets()) {
    ReverseNested resellerToProduct = bucket.getAggregations().g
et("reseller_to_product");
    resellerToProduct.getDocCount(); // Doc count
}
```

Children Aggregation

> 一种特殊的单桶聚合，可以将父文档类型上的桶聚合到子文档上。

下面是如何使用 `Java API` 使用Children Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
    AggregationBuilders
        .children("agg", "reseller"); // agg 是聚合名，reseller 是
子类型
```

使用聚合请求

```
import org.elasticsearch.join.aggregations.Children;
```

```
// sr is here your SearchResponse object
Children agg = sr.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```

Terms Aggregation 词元聚合

> 基于某个field，该 field 内的每一个【唯一词元】为一个桶，并计算每个桶内文档个数。默认返回顺序是按照文档个数多少排序。当不返回所有 buckets 的情况，文档个数可能不准确。

下面是如何使用 `Java API` 使用Terms Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilders
    .terms("genders")
    .field("gender");
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
```

```
// sr is here your SearchResponse object
Terms genders = sr.getAggregations().get("genders");

// For each entry
for (Terms.Bucket entry : genders.getBuckets()) {
    entry.getKey();      // Term
    entry.getDocCount(); // Doc count
}
```

Order 排序

通过 `doc_count` 按升序排列：

```
AggregationBuilders
    .terms("genders")
    .field("gender")
    .order(Terms.Order.count(true))
```

按字词顺序，升序排列：

```
AggregationBuilders
    .terms("genders")
    .field("gender")
    .order(Terms.Order.term(true))
```

按metrics 子聚合排列（标示为聚合名）

```
AggregationBuilders
    .terms("genders")
    .field("gender")
    .order(Terms.Order.aggregation("avg_height", false))
    .subAggregation(
        AggregationBuilders.avg("avg_height").field("height")
    )
```

Significant Terms Aggregation

> 返回集合中感兴趣的或者不常见的词条的聚合

下面是如何使用 `Java API` 使用Significant Terms Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
        AggregationBuilders
                .significantTerms("significant_countries")
                .field("address.country");

// Let say you search for men only
SearchResponse sr = client.prepareSearch()
        .setQuery(QueryBuilders.termQuery("gender", "male"))
        .addAggregation(aggregation)
        .get();
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.significant.
SignificantTerms;
```

```
// sr is here your SearchResponse object
SignificantTerms agg = sr.getAggregations().get("significant_cou
ntries");

// For each entry
for (SignificantTerms.Bucket entry : agg.getBuckets()) {
    entry.getKey();      // Term
    entry.getDocCount(); // Doc count
}
```

Range Aggregation 范围聚合

> 基于某个值（可以是 field 或 script），以【字段范围】来桶分聚合。范围聚合包括 from 值，不包括 to 值（区间前闭后开）。

下面是如何使用 `Java API` 使用Range Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
        AggregationBuilders
                .range("agg")
                .field("height")
                .addUnboundedTo(1.0f)               // from -inf
inity to 1.0 (excluded)
                .addRange(1.0f, 1.5f)               // from 1.0
to 1.5 (excluded)
                .addUnboundedFrom(1.5f);            // from 1.5
to +infinity
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();             // Range as
 key
    Number from = (Number) entry.getFrom();          // Bucket f
rom
    Number to = (Number) entry.getTo();              // Bucket t
o
    long docCount = entry.getDocCount();    // Doc count

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]",
key, from, to, docCount);
}
```

输出：

```
key [*-1.0], from [-Infinity], to [1.0], doc_count [9]
key [1.0-1.5], from [1.0], to [1.5], doc_count [21]
key [1.5-*], from [1.5], to [Infinity], doc_count [20]
```

Date Range Aggregation 日期范围聚合

> 日期范围聚合——基于日期类型的值，以【日期范围】来桶分聚合。
>
> 日期范围可以用各种 Date Math 表达式。
>
> 同样的，包括 from 的值，不包括 to 的值。

下面是如何使用 `Java API` 使用Date Range Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
        AggregationBuilders
                .dateRange("agg")
                .field("dateOfBirth")
                .format("yyyy")
                .addUnboundedTo("1950")    // from -infinity to
1950 (excluded)
                .addRange("1950", "1960")  // from 1950 to 1960
(excluded)
                .addUnboundedFrom("1960"); // from 1960 to +infi
nity
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();             // Date
range as key
    DateTime fromAsDate = (DateTime) entry.getFrom();   // Date
bucket from as a Date
    DateTime toAsDate = (DateTime) entry.getTo();       // Date
bucket to as a Date
    long docCount = entry.getDocCount();                // Doc c
ount

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]",
key, fromAsDate, toAsDate, docCount);
}
```

输出：

```
key [*-1950], from [null], to [1950-01-01T00:00:00.000Z], doc_co
unt [8]
key [1950-1960], from [1950-01-01T00:00:00.000Z], to [1960-01-01
T00:00:00.000Z], doc_count [5]
key [1960-*], from [1960-01-01T00:00:00.000Z], to [null], doc_co
unt [37]
```

Ip Range Aggregation Ip范围聚合

下面是如何使用 `Java API` 使用Ip Range Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregatorBuilder<?> aggregation =
        AggregationBuilders
                .ipRange("agg")
                .field("ip")
                .addUnboundedTo("192.168.1.0")             // fr
om -infinity to 192.168.1.0 (excluded)
                .addRange("192.168.1.0", "192.168.2.0")    // fr
om 192.168.1.0 to 192.168.2.0 (excluded)
                .addUnboundedFrom("192.168.2.0");          // fr
om 192.168.2.0 to +infinity
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();            // Ip range
as key
    String fromAsString = entry.getFromAsString();  // Ip bucket
 from as a String
    String toAsString = entry.getToAsString();      // Ip bucket
 to as a String
    long docCount = entry.getDocCount();            // Doc count

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]",
key, fromAsString, toAsString, docCount);
}
```

输出：

```
key [*-1950], from [null], to [1950-01-01T00:00:00.000Z], doc_co
unt [8]
key [1950-1960], from [1950-01-01T00:00:00.000Z], to [1960-01-01
T00:00:00.000Z], doc_count [5]
key [1960-*], from [1960-01-01T00:00:00.000Z], to [null], doc_co
unt [37]
```

Histogram Aggregation 直方图聚合

> 基于文档中的某个【数值类型】字段，通过计算来动态的分桶。

下面是如何使用 `Java API` 使用Histogram Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
        AggregationBuilders
                .histogram("agg")
                .field("height")
                .interval(1);
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.histogram.Hi
stogram;
```

```
// sr is here your SearchResponse object
Histogram agg = sr.getAggregations().get("agg");

// For each entry
for (Histogram.Bucket entry : agg.getBuckets()) {
    Number key = (Number) entry.getKey();   // Key
    long docCount = entry.getDocCount();    // Doc count

    logger.info("key [{}], doc_count [{}]", key, docCount);
}
```

Date Histogram Aggregation 日期范围直方图聚合

> 与直方图类似的多bucket聚合，但只能应用于日期值.。

下面是如何使用 `Java API` 使用 Date Histogram Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
        AggregationBuilders
                .dateHistogram("agg")
                .field("dateOfBirth")
                .dateHistogramInterval(DateHistogramInterval.YEA
R);
```

或者把时间间隔设置为10天

```
AggregationBuilder aggregation =
        AggregationBuilders
                .dateHistogram("agg")
                .field("dateOfBirth")
                .dateHistogramInterval(DateHistogramInterval.day
s(10));
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.histogram.Hi
stogram;
```

```
// sr is here your SearchResponse object
Histogram agg = sr.getAggregations().get("agg");

// For each entry
for (Histogram.Bucket entry : agg.getBuckets()) {
    DateTime key = (DateTime) entry.getKey();    // Key
    String keyAsString = entry.getKeyAsString(); // Key as Strin
g
    long docCount = entry.getDocCount();         // Doc count

    logger.info("key [{}], date [{}], doc_count [{}]", keyAsStri
ng, key.getYear(), docCount);
}
```

输出：

```
key [1942-01-01T00:00:00.000Z], date [1942], doc_count [1]
key [1945-01-01T00:00:00.000Z], date [1945], doc_count [1]
key [1946-01-01T00:00:00.000Z], date [1946], doc_count [1]
...
key [2005-01-01T00:00:00.000Z], date [2005], doc_count [1]
key [2007-01-01T00:00:00.000Z], date [2007], doc_count [2]
key [2008-01-01T00:00:00.000Z], date [2008], doc_count [3]
```

Geo Distance Aggregation 地理距离聚合

> 在geo_point字段上工作的多bucket聚合和概念上的工作非常类似于range(范围)聚合.用户可以定义原点的点和距离范围的集合。聚合计算每个文档值与原点的距离，并根据范围确定其所属的bucket(桶)（如果文档和原点之间的距离落在bucket(桶)的距离范围内，则文档属于bucket(桶) ）

下面是如何使用 `Java API` 使用 Geo Distance Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
        AggregationBuilders
                .geoDistance("agg", new GeoPoint(48.84237171118314,2.33320027692004))
                .field("address.location")
                .unit(DistanceUnit.KILOMETERS)
                .addUnboundedTo(3.0)
                .addRange(3.0, 10.0)
                .addRange(10.0, 500.0);
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

```
// sr is here your SearchResponse object
Range agg = sr.getAggregations().get("agg");

// For each entry
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();    // key as String
    Number from = (Number) entry.getFrom(); // bucket from value
    Number to = (Number) entry.getTo();     // bucket to value
    long docCount = entry.getDocCount();    // Doc count

    logger.info("key [{}], from [{}], to [{}], doc_count [{}]", key, from, to, docCount);
}
```

输出：

```
key [*-3.0], from [0.0], to [3.0], doc_count [161]
key [3.0-10.0], from [3.0], to [10.0], doc_count [460]
key [10.0-500.0], from [10.0], to [500.0], doc_count [4925]
```

Geo Hash Grid Aggregation GeoHash网格聚合

> 在geo_point字段和组上工作的多bucket聚合将指向网格中表示单元格的bucket。生成的网格可以是稀疏的，并且只包含具有匹配数据的单元格。每个单元格使用具有用户可定义精度的 geohash 进行标记。

下面是如何使用 `Java API` 使用Geo Hash Grid Aggregation

准备聚合请求

下面是如何创建聚合请求的是示例：

```
AggregationBuilder aggregation =
        AggregationBuilders
                .geohashGrid("agg")
                .field("address.location")
                .precision(4);
```

使用聚合请求

```
import org.elasticsearch.search.aggregations.bucket.geogrid.GeoHashGrid;
```

```
// sr is here your SearchResponse object
GeoHashGrid agg = sr.getAggregations().get("agg");

// For each entry
for (GeoHashGrid.Bucket entry : agg.getBuckets()) {
    String keyAsString = entry.getKeyAsString(); // key as String
    GeoPoint key = (GeoPoint) entry.getKey();    // key as geo point
    long docCount = entry.getDocCount();         // Doc count

    logger.info("key [{}], point {}, doc_count [{}]", keyAsString, key, docCount);
}
```

输出：

``` key [gbqu], point [47.197265625, -1.58203125], doc_count [1282] key [gbvn], point [50.361328125, -4.04296875], doc_count [1248] key [u1j0], point [50.712890625, 7.20703125], doc_count [1156] key [u0j2], point [45.087890625, 7.55859375], doc_count [1138] ...
```

Query DSL

> Elasticsearch 提供了一个基于 JSON 的完整的查询 DSL 来定义查询。

Elasticsearch以类似于REST Query DSL 的方式提供完整的Java查询dsl。 查询构建器的工厂是 `QueryBuilders` 。 一旦的查询准备就绪，就可以使用Search API 。

要使用QueryBuilder，只需将它们导入到类中：

```
import static org.elasticsearch.index.query.QueryBuilders.*;
```

> 注意，可以使用 `QueryBuilder` 对象上的 `toString()` 方法打印。

`QueryBuilder` 可以用于接受任何查询API，如 `count` 和 `search` 。

Match All Query

> 最简单的查询，它匹配所有文档

查看 Match All Query

```
QueryBuilder qb = matchAllQuery();
```

Full text queries 全文搜索

高级别的全文搜索通常用于在全文字段（例如：一封邮件的正文）上进行全文搜索。它们了解如何分析查询的字段，并在执行之前将每个字段的分析器（或搜索分析器）应用于查询字符串。

这样的查询有以下这些：

- 匹配查询（match query）

用于执行全文查询的标准查询，包括模糊匹配和词组或邻近程度的查询

查看 Match Query

```
QueryBuilder qb = matchQuery(
    "name",                  //field 字段
    "kimchy elasticsearch"   // text
);
```

- 多字段查询（multi_match query）

可以用来对多个字段的版本进行匹配查询

查看 Multi Match Query

```
QueryBuilder qb = multiMatchQuery(
    "kimchy elasticsearch", //text
    "user", "message"       //fields 多个字段
);
```

- 常用术语查询（common_terms query）

可以对一些比较专业的偏门词语进行的更加专业的查询

查看Common Terms Query

```
QueryBuilder qb = commonTermsQuery("name",    //field 字段
                                   "kimchy");  // value
```

- 查询语句查询（query_string query）

与lucene查询语句的语法结合的更加紧密的一种查询，允许你在一个查询语句中使用多个 特殊条件关键字（如：AND|OR|NOT ）对多个字段进行查询，当然这种查询仅限 `专家用户` 去使用。

查看Query String Query

```
QueryBuilder qb = queryStringQuery("+kimchy -elasticsearch");
//text
```

- 简单查询语句（simple_query_string）

是一种适合直接暴露给用户的简单的且具有非常完善的查询语法的查询语句

查看Simple Query String Query

```
QueryBuilder qb = simpleQueryStringQuery("+kimchy -elasticsearch
");    //text
```

Term level queries 术语查询

虽然全文查询将在执行之前分析查询字符串，但是项级别查询对存储在反向索引中的确切项进行操作。

通常用于结构化数据，如数字、日期和枚举，而不是全文字段。或者，在分析过程之前，它允许你绘制低级查询。

这样的查询有以下这些：

- Term Query（项查询）

查询包含在指定字段中指定的确切值的文档。

查看Term Query

```
QueryBuilder qb = termQuery(
    "name",    //field
    "kimchy"   //text
);
```

- Terms Query（多项查询）

查询包含任意一个在指定字段中指定的多个确切值的文档。

查看Terms Query

```
QueryBuilder qb = termsQuery("tags",    //field
    "blue", "pill");                    //values
```

- Range Query（范围查询）

查询指定字段包含指定范围内的值（日期，数字或字符串）的文档。

查看Range Query

方法：

1. gte() :范围查询将匹配字段值大于或等于此参数值的文档。

2. gt() :范围查询将匹配字段值大于此参数值的文档。
3. lte() :范围查询将匹配字段值小于或等于此参数值的文档。
4. lt() :范围查询将匹配字段值小于此参数值的文档。
5. from() 开始值 to() 结束值 这两个函数与includeLower()和includeUpper()函数配套使用。
6. includeLower(true) 表示 from() 查询将匹配字段值大于或等于此参数值的文档。
7. includeLower(false) 表示 from() 查询将匹配字段值大于此参数值的文档。
8. includeUpper(true) 表示 to() 查询将匹配字段值小于或等于此参数值的文档。
9. includeUpper(false) 表示 to() 查询将匹配字段值小于此参数值的文档。

```
QueryBuilder qb = rangeQuery("price")   //field
    .from(5)                            //开始值，与includeLower()和includeUpper()函数配套使用
    .to(10)                             //结束值，与includeLower()和includeUpper()函数配套使用
    .includeLower(true)                 // true: 表示 from() 查询将匹配字段值大于或等于此参数值的文档;   false:表示 from() 查询将匹配字段值大于此参数值的文档。
    .includeUpper(false);               //true:表示 to() 查询将匹配字段值小于或等于此参数值的文档;   false:表示 to() 查询将匹配字段值小于此参数值的文档。
```

实例

```
// Query
RangeQueryBuilder rangeQueryBuilder = QueryBuilders.rangeQuery("
age");
rangeQueryBuilder.from(19);
rangeQueryBuilder.to(21);
rangeQueryBuilder.includeLower(true);
rangeQueryBuilder.includeUpper(true);
//RangeQueryBuilder rangeQueryBuilder = QueryBuilders.rangeQuery
("age").gte(19).lte(21);
// Search
SearchRequestBuilder searchRequestBuilder = client.prepareSearch
(index);
searchRequestBuilder.setTypes(type);
searchRequestBuilder.setQuery(rangeQueryBuilder);
// 执行
SearchResponse searchResponse = searchRequestBuilder.execute().a
ctionGet();
```

上面代码中的查询语句与下面的是等价的：

```
QueryBuilder queryBuilder = QueryBuilders.rangeQuery("age").gte(
19).lte(21);
```

- Exists Query（存在查询）

查询指定的字段包含任何非空值的文档,如果指定字段上至少存在一个no-null的值就会返回该文档。

查看Exists Query

```
QueryBuilder qb = existsQuery("name");
```

实例

```
// Query
ExistsQueryBuilder existsQueryBuilder = QueryBuilders.existsQuer
y("name");
// Search
SearchRequestBuilder searchRequestBuilder = client.prepareSearch
(index);
searchRequestBuilder.setTypes(type);
searchRequestBuilder.setQuery(existsQueryBuilder);
// 执行
SearchResponse searchResponse = searchRequestBuilder.get();
```

举例说明，下面的几个文档都会得到上面代码的匹配：

```
{ "name": "yoona" }
{ "name": "" }
{ "name": "-" }
{ "name": ["yoona"] }
{ "name": ["yoona", null ] }
```

第一个是字符串，是一个非null的值。

第二个是空字符串，也是非null。

第三个使用标准分析器的情况下尽管不会返回词条，但是原始字段值是非null的（Even though the standard analyzer would emit zero tokens, the original field is non-null）。

第五个中至少有一个是非null值。

下面几个文档不会得到上面代码的匹配：

```
{ "name": null }
{ "name": [] }
{ "name": [null] }
{ "user":  "bar" }
```

第一个是null值。

第二个没有值。

第三个只有null值，至少需要一个非null值。

第四个与指定字段不匹配。

- Prefix Query（前缀查询）

查找指定字段包含以指定的精确前缀开头的值的文档。

查看Prefix Query

```
QueryBuilder qb = prefixQuery(
    "brand",    //field
    "heine"     //prefix
);
```

- Wildcard Query（通配符查询）

查询指定字段包含与指定模式匹配的值的文档，其中该模式支持单字符通配符（？）和多字符通配符（*）,和前缀查询一样，通配符查询指定字段是未分析的（not analyzed）

可以使用星号代替0个或多个字符，使用问号代替一个字符。星号表示匹配的数量不受限制，而后者的匹配字符数则受到限制。这个技巧主要用于英文搜索中，如输入““computer*”，就可以找到“computer、computers、computerised、computerized”等单词，而输入“comp?ter”，则只能找到“computer、compater、competer”等单词。注意的是通配符查询不太注重性能，在可能时尽量避免，特别是要避免前缀通配符（以通配符开始的词条）。

```
QueryBuilder qb = wildcardQuery("user", "k?mc*");
```

实例

```
// Query
WildcardQueryBuilder wildcardQueryBuilder = QueryBuilders.wildca
rdQuery("country", "西*牙");
// Search
SearchRequestBuilder searchRequestBuilder = client.prepareSearch
(index);
searchRequestBuilder.setTypes(type);
searchRequestBuilder.setQuery(wildcardQueryBuilder);
// 执行
SearchResponse searchResponse = searchRequestBuilder.get();
```

查看Wildcard Query

- Regexp Query（正则表达式查询）

查询指定的字段包含与指定的正则表达式匹配的值的文档。

和前缀查询一样，正则表达式查询指定字段是未分析的（not analyzed）。正则表达式查询的性能取决于所选的正则表达式。如果我们的正则表达式匹配许多词条，查询将很慢。一般规则是，正则表达式匹配的词条数越高，查询越慢。

查看Regexp Query

```
QueryBuilder qb = regexpQuery(
    "name.first",        //field
    "s.*y");             //regexp
```

实例

```
// Query
RegexpQueryBuilder regexpQueryBuilder = QueryBuilders.regexpQuer
y("country", "(西班|葡萄)牙");

// Search
SearchRequestBuilder searchRequestBuilder = client.prepareSearch
(index);
searchRequestBuilder.setTypes(type);
searchRequestBuilder.setQuery(regexpQueryBuilder);
// 执行
SearchResponse searchResponse = searchRequestBuilder.get();
```

- Fuzzy Query（模糊查询）

查询指定字段包含与指定术语模糊相似的术语的文档。模糊性测量为1或2的Levenshtein。

如果指定的字段是string类型，模糊查询是基于编辑距离算法来匹配文档。编辑距离的计算基于我们提供的查询词条和被搜索文档。如果指定的字段是数值类型或者日期类型，模糊查询基于在字段值上进行加减操作来匹配文档（The fuzzy query uses similarity based on Levenshtein edit distance for string fields, and a +/- margin on numeric and date fields）。此查询很占用CPU资源，但当需要模糊匹配时它很有用，例如，当用户拼写错误时。另外我们可以在搜索词的尾部加上字符 “~” 来进行模糊查询。

查看Fuzzy Query

```
QueryBuilder qb = fuzzyQuery(
    "name",     //field
    "kimzhy"    //text
);
```

- Type Query（类型查询）

查询指定类型的文档。

查看Type Query

```
QueryBuilder qb = typeQuery("my_type"); //type
```

- Ids Query（ID查询）

查询具有指定类型和 ID 的文档。

查看Ids Query

```
QueryBuilder qb = idsQuery("my_type", "type2")
    .addIds("1", "4", "100");

QueryBuilder qb = idsQuery()  // type 是可选择的，可以不写
    .addIds("1", "4", "100");
```

Compound queries

复合查询用来包装其他复合或者叶子查询，一方面可综合其结果和分数，从而改变它的行为，另一方面可从查询切换到过滤器上下文。此类查询包含：

- constant_score 查询

这是一个包装其他查询的查询，并且在过滤器上下文中执行。与此查询匹配的所有文件都需要返回相同的“常量” _score 。

查看Constant Score Query

```
QueryBuilder qb = constantScoreQuery(
        termQuery("name","kimchy")      //查询语句
    )
    .boost(2.0f);   //分数
```

- bool 查询

组合多个叶子并发查询或复合查询条件的默认查询类型，例如must, should, must_not, 以及 filter 条件。 在 must 和 should 子句他们的分数相结合-匹配条件越多，预期越好-而 must_not 和 filter 子句在过滤器上下文中执行。

查看Bool Query

```
QueryBuilder qb = boolQuery()
    .must(termQuery("content", "test1"))    //must query
    .must(termQuery("content", "test4"))
    .mustNot(termQuery("content", "test2")) //must not query
    .should(termQuery("content", "test3"))  // should query
    .filter(termQuery("content", "test5// 与一般查询作用一样，只不过
不参与评分
```

- dis_max 查询

支持多并发查询的查询，并可返回与任意查询条件子句匹配的任何文档类型。与 bool 查询可以将所有匹配查询的分数相结合使用的方式不同的是，dis_max 查询只使用最佳匹配查询条件的分数。

查看Dis Max Query

```
QueryBuilder qb = disMaxQuery()
    .add(termQuery("name", "kimchy"))
    .add(termQuery("name", "elasticsearch"))
    .boost(1.2f)                   //boost factor
    .tieBreaker(0.7f);   //tie breaker
```

- function_score 查询

使用函数修改主查询返回的分数，以考虑诸如流行度，新近度，距离或使用脚本实现的自定义算法等因素。

查看Function Score Query

```
import static org.elasticsearch.index.query.functionscore.ScoreF
unctionBuilders.*;
```

```
FilterFunctionBuilder[] functions = {
        new FunctionScoreQueryBuilder.FilterFunctionBuilder(
                matchQuery("name", "kimchy"),                //
根据查询添加第一个function
                randomFunction("ABCDEF")),                   //
根据给定的种子随机分数
        new FunctionScoreQueryBuilder.FilterFunctionBuilder(
                exponentialDecayFunction("age", 0L, 1L))     //
根据年龄字段添加另一个function

};
QueryBuilder qb = QueryBuilders.functionScoreQuery(functions);
```

- boosting 查询

返回匹配 positive 查询的文档，并且当减少文档的分数时其结果也匹配 negative 查询。

查看Boosting Query

```
QueryBuilder qb = boostingQuery(
        termQuery("name","kimchy"),
        termQuery("name","dadoonet"))
    .negativeBoost(0.2f);
```

- indices 查询

对指定的索引执行一个查询，对其他索引执行另一个查询。

查看Indices Query

> 在5.0.0中已弃用。用搜索 _index 字段来代替

```
// Using another query when no match for the main one
QueryBuilder qb = indicesQuery(
        termQuery("tag", "wow"),
        "index1", "index2"
    ).noMatchQuery(termQuery("tag", "kow"));
```

```
// Using all (match all) or none (match no documents)
QueryBuilder qb = indicesQuery(
        termQuery("tag", "wow"),
        "index1", "index2"
    ).noMatchQuery("all");
```

Joining queries

在像 ElasticSearch 这样的分布式系统中执行全 SQL 风格的连接查询代价昂贵，是不可行的。相应地，为了实现水平规模地扩展，ElasticSearch 提供了两种形式的 join。

- nested query (嵌套查询)

文档中可能包含嵌套类型的字段，这些字段用来索引一些数组对象，每个对象都可以作为一条独立的文档被查询出来(用嵌套查询)

查看Nested Query

```
QueryBuilder qb = nestedQuery(
        "obj1",                       //nested 嵌套文档的路径
        boolQuery()                   // 查询 查询中引用的任何字段都必须使用完整路径（fully qualified）。
                .must(matchQuery("obj1.name", "blue"))
                .must(rangeQuery("obj1.count").gt(5)),
        ScoreMode.Avg                 // score 模型 ScoreMode.Max, ScoreMode.Min, ScoreMode.Total, ScoreMode.Avg or ScoreMode.None
    );
```

- has_child (有子查询) and has_parent (有父查询) queries

一类父子关系可以存在单个的索引的两个类型的文档之间。has_child 查询将返回其子文档能满足特定的查询的父文档，而 has_parent 则返回其父文档能满足特定查询的子文档

Has Child Query

查看Has Child Query

使用 `has_child` 查询时，必须使用 `PreBuiltTransportClient` 而不是常规 `Client`，这个点很重要：

```
Settings settings = Settings.builder().put("cluster.name", "elasticsearch").build();
TransportClient client = new PreBuiltTransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress(new InetSocketAddress(InetAddresses.forString("127.0.0.1"), 9300)));
```

否则， `parent-join` 模块不会被加载，并且不能从transport client 使用 `has_child` 查询。

```
QueryBuilder qb = JoinQueryBuilders.hasChildQuery(
    "blog_tag",                     //要查询的子类型
    termQuery("tag","something"),   //查询
    ScoreMode.Avg                   //score 模型 ScoreMode.Avg, ScoreMode.Max, ScoreMode.Min, ScoreMode.None or ScoreMode.Total
);
```

Has Parent Query

查看Has Parent

使用 `has_parent` 查询时，必须使用 `PreBuiltTransportClient` 而不是常规 `Client` ，这个点很重要：

```
Settings settings = Settings.builder().put("cluster.name", "elasticsearch").build();
TransportClient client = new PreBuiltTransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress(new InetSocketAddress(InetAddresses.forString("127.0.0.1"), 9300)));
```

否则， `parent-join` 模块不会被加载，并且不能从transport client 使用 `has_child` 查询。

```
QueryBuilder qb = JoinQueryBuilders.hasParentQuery(
    "blog",                         //要查询的子类型
    termQuery("tag","something"),   //查询
    false                           //是否从父hit的score 传给子 hit
);
```

参考 term 查询中的terms-lookup mechanism ，它允许你在另一个文档的值中创建一个term 查询。

Geo queries 地理位置查询

Elasticsearch支持两种类型的地理数据：geo_point类型支持成对的纬度/经度，geo_shape类型支持点、线、圆、多边形、多个多边形等。 在这组的查询中：

- geo_shape查询

查找要么相交，包含的，要么指定形状不相交的地理形状的文档。

查看Geo Shape Query

> `geo_shape` 类型使用 `Spatial4J` 和 `JTS` ，这两者都是可选的依赖项。因此，必须将 `Spatial4J` 和 `JTS` 添加到 `classpath` 中才能使用此类型：

```
<dependency>
    <groupId>org.locationtech.spatial4j</groupId>
    <artifactId>spatial4j</artifactId>
    <version>0.6</version>
</dependency>

<dependency>
    <groupId>com.vividsolutions</groupId>
    <artifactId>jts</artifactId>
    <version>1.13</version>
    <exclusions>
        <exclusion>
            <groupId>xerces</groupId>
            <artifactId>xercesImpl</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

```
// Import ShapeRelation and ShapeBuilder
import org.elasticsearch.common.geo.ShapeRelation;
import org.elasticsearch.common.geo.builders.ShapeBuilder;
```

```
List<Coordinate> points = new ArrayList<>();
points.add(new Coordinate(0, 0));
points.add(new Coordinate(0, 10));
points.add(new Coordinate(10, 10));
points.add(new Coordinate(10, 0));
points.add(new Coordinate(0, 0));

QueryBuilder qb = geoShapeQuery(
        "pin.location",                         //field
        ShapeBuilders.newMultiPoint(points)     //shape
        .relation(ShapeRelation.WITHIN);        //relation 可以是
 ShapeRelation.CONTAINS, ShapeRelation.WITHIN, ShapeRelation.INT
ERSECTS 或 ShapeRelation.DISJOINT
```

```
// Using pre-indexed shapes
QueryBuilder qb = geoShapeQuery(
        "pin.location",                  //field
        "DEU",                           //The ID of the documen
t that containing the pre-indexed shape.
        "countries")                     //Index type where the
pre-indexed shape is.
        .relation(ShapeRelation.WITHIN)) //relation
        .indexedShapeIndex("shapes")     //Name of the index whe
re the pre-indexed shape is. Defaults to shapes.
        .indexedShapePath("location");   //The field specified a
s path containing the pre-indexed shape. Defaults to shape.
```

- geo_bounding_box 查询

查找落入指定的矩形地理点的文档。

查看Geo Bounding Box Query

```
QueryBuilder qb = geoBoundingBoxQuery("pin.location") //field
    .setCorners(40.73, -74.1,                         //bounding
 box top left point
                40.717, -73.99);                      //bounding
 box bottom right point
```

- geo_distance 查询

查找在一个中心点指定范围内的地理点文档。

查看Geo Distance Query

```
QueryBuilder qb = geoDistanceQuery("pin.location")  //field
    .point(40, -70)                                 //center poi
nt
    .distance(200, DistanceUnit.KILOMETERS);        //distance f
rom center point
```

- geo_polygon 查询

查找指定多边形内地理点的文档。

查看Geo Polygon Query

```
List<GeoPoint> points = new ArrayList<>();             //add you
r polygon of points a document should fall within
points.add(new GeoPoint(40, -70));
points.add(new GeoPoint(30, -80));
points.add(new GeoPoint(20, -90));

QueryBuilder qb =
        geoPolygonQuery("pin.location", points);       //initial
ise the query with field and points
```

Specialized queries

- more_like_this query(相似度查询)

这个查询能检索到与指定文本、文档或者文档集合相似的文档。

查看More Like This Query

```
String[] fields = {"name.first", "name.last"};                 /
/fields
String[] texts = {"text like this one"};                       /
/text
Item[] items = null;

QueryBuilder qb = moreLikeThisQuery(fields, texts, items)
    .minTermFreq(1)                                            /
/ignore threshold
    .maxQueryTerms(12);                                        /
/max num of Terms in generated queries
```

- script query

该查询允许脚本充当过滤器。 另请参阅 function_score query 。

查看Script Query

```
QueryBuilder qb = scriptQuery(
    new Script("doc['num1'].value > 1") //inlined script
);
```

如果已经在每个数据节点上存储名为 `myscript.painless 的脚本，请执行以下操作：

```
doc['num1'].value > params.param1
```

然后使用：

```
QueryBuilder qb = scriptQuery(
    new Script(
        ScriptType.FILE,                       //脚本类型 ScriptT
ype.FILE, ScriptType.INLINE， ScriptType.INDEXED
        "painless",                            //Scripting engin
e 脚本引擎
        "myscript",                            //Script name 脚本
名
        Collections.singletonMap("param1", 5)) //Parameters as a
 Map of <String, Object>
);
```

- Percolate Query

查看Percolate Query

```
Settings settings = Settings.builder().put("cluster.name", "elas
ticsearch").build();
TransportClient client = new PreBuiltTransportClient(settings);
client.addTransportAddress(new InetSocketTransportAddress(new In
etSocketAddress(InetAddresses.forString("127.0.0.1"), 9300)));
```

在可以使用 `percolate` 查询之前，应该添加 `percolator` 映射，并且应该对包含 `percolator` 查询的文档建立索引：

```
// create an index with a percolator field with the name 'query'
:
client.admin().indices().prepareCreate("myIndexName")
                        .addMapping("query", "query", "type=perc
olator")
                        .addMapping("docs", "content", "type=tex
t")
                        .get();

//This is the query we're registering in the percolator
QueryBuilder qb = termQuery("content", "amazing");

//Index the query = register it in the percolator
client.prepareIndex("myIndexName", "query", "myDesignatedQueryNa
me")
    .setSource(jsonBuilder()
        .startObject()
            .field("query", qb) // Register the query
        .endObject())
    .setRefreshPolicy(RefreshPolicy.IMMEDIATE) // Needed when th
e query shall be available immediately
    .get();
```

在上面的index中query名为 `myDesignatedQueryName`

为了检查文档注册查询,使用这个代码:

```
//Build a document to check against the percolator
XContentBuilder docBuilder = XContentFactory.jsonBuilder().start
Object();
docBuilder.field("content", "This is amazing!");
docBuilder.endObject(); //End of the JSON root object

PercolateQueryBuilder percolateQuery = new PercolateQueryBuilder
("query", "docs", docBuilder.bytes());

// Percolate, by executing the percolator query in the query dsl
:
SearchResponse response = client().prepareSearch("myIndexName")
        .setQuery(percolateQuery))
        .get();
//Iterate over the results
for(SearchHit hit : response.getHits()) {
    // Percolator queries as hit
}
```

Span queries

- span_term查询

等同于 term query ，但与其他Span查询一起使用。

查看 Span Term Query

```
QueryBuilder qb = spanTermQuery(
    "user",                                     //field
    "kimchy"                                    //value
);
```

- span_multi查询

包含term, range, prefix, wildcard, regexp 或者 fuzzy 查询。

查看Span Multi Term Query

```
QueryBuilder qb = spanMultiTermQueryBuilder(
    prefixQuery("user", "ki")                   //可以是MultiTerm
QueryBuilder 的 扩展  比如：FuzzyQueryBuilder, PrefixQueryBuilder,
 RangeQueryBuilder, RegexpQueryBuilder ， WildcardQueryBuilder。
);
```

- span_first查询

接受另一个Span查询，其匹配必须出现在字段的前N个位置。

查看Span First Query

```
QueryBuilder qb = spanFirstQuery(
    spanTermQuery("user", "kimchy"),            //query
    3                                           //max end positi
on
);
```

- span_near查询

接受多个Span查询，其匹配必须在彼此的指定距离内，并且可能顺序相同。

查看Span Near Query

```
QueryBuilder qb = spanNearQuery(
    spanTermQuery("field","value1"),            //span term quer
ies
    12)                                         //slop factor: t
he maximum number of intervening unmatched positions
    .addClause(spanTermQuery("field","value2")) //span term quer
ies
    .addClause(spanTermQuery("field","value3")) //span term quer
ies
    .inOrder(false);                            //whether matche
s are required to be in-order
```

- span_or查询

组合多个Span查询 - 返回与任何指定查询匹配的文档。

查看Span Or Query

```
QueryBuilder qb = spanOrQuery(
    spanTermQuery("field","value1"))
    .addClause(spanTermQuery("field","value2"))
    .addClause(spanTermQuery("field","value3"));   //span term q
ueries
```

- span_not查询

包装另一个Span查询，并排除与该查询匹配的所有文档。

查看Span Not Query

```
QueryBuilder qb = spanNotQuery(
    spanTermQuery("field","value1"),   //span query whose matche
s are filtered
    spanTermQuery("field","value2"));  //span query whose matche
s must not overlap those returned
```

- span_containing 查询

接受Span查询的列表，但仅返回与第二个Spans查询匹配的Span。

查看 Span Containing Query

```
QueryBuilder qb = spanContainingQuery(
    spanNearQuery(spanTermQuery("field1","bar"), 5) //big part
          .addClause(spanTermQuery("field1","baz"))
          .inOrder(true),
    spanTermQuery("field1","foo"));                 //little par
t
```

- span_within查询

只要其 span 位于由其他Span查询列表返回的范围内，就会返回单个Span查询的结果，

查看Span Within Query

```
QueryBuilder qb = spanWithinQuery(
    spanNearQuery(spanTermQuery("field1", "bar"), 5)  //big part
        .addClause(spanTermQuery("field1", "baz"))
        .inOrder(true),
    spanTermQuery("field1", "foo"));                  //little p
art
```